HoGent

BEDRIJF
EN
ORGANISATIE

Factory pattern

Creatie van klassen

Bakken op OO niveau : pizza's

Voorbeeld "pizzeria"

- Stel je runt een pizzeria en als vooruitstrevende eigenaar, schrijf je de volgende code:

```
public Pizza orderPizza(String type)
{
        Pizza pizza;
        switch (type.toLowerCase())
        {
                case "cheese": pizza = new CheesePizza(); break;
                case "greek": pizza = new GreekPizza(); break;
                case "pepperoni": pizza = new PepperoniPizza();break;
                default: pizza = null;
        }
        if (pizza != null){
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
       }
        return pizza;
}
```

Voorbeeld “pizzeria”

- Je dient de concurrentie te volgen. Je voegt twee nieuwe pizza’s "clam" en "veggie" toe en je haalt "greek" van het menu

```
public Pizza orderPizza(String type)
{
        Pizza pizza;
        switch (type.toLowerCase())
        {
                case "cheese": pizza = new CheesePizza(); break;
                case "greek": pizza = new GreekPizza(); break;
                case "pepperoni": pizza = new PepperoniPizza();break;
                case "clam": pizza = new ClamPizza(); break;
                case "veggie": pizza = new VeggiePizza(); break;
                default: pizza = null;
        }
        if (pizza != null){
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
        }
        return pizza;
}
```

Deze code is NIET GESLOTEN voor VERANDERING. Als de pizzeria zijn pizza-aanbiedingen verandert, dan moeten we in de code duiken en deze veranderen.

De objectcreatie isoleren

```
public Pizza orderPizza(String type)
{
        Pizza pizza;

        if (pizza != null){
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();
        }
        return pizza;
}
```


```
switch (type.toLowerCase())
{
        case "cheese": pizza = new CheesePizza(); break;
        case "pepperoni": pizza = new PepperoniPizza();break;
        case "clam": pizza = new ClamPizza(); break;
        case "veggie": pizza = new VeggiePizza(); break;
        default: pizza = null;
}
```

We pakken de code voor de creatie op en verplaatsen deze naar een ander object dat alleen maar het maken van pizza’s als taak zal hebben. Dit object noemen we Factory.

Een eenvoudige pizzafabriek bouwen

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        switch (type.toLowerCase()) {
            case "cheese":
                return new CheesePizza();
            case "pepperoni":
                return new PepperoniPizza();
            case "clam":
                return new ClamPizza();
            case "veggie":
                return new VeggiePizza();
            default:
                return null;
        }
    }
}
```

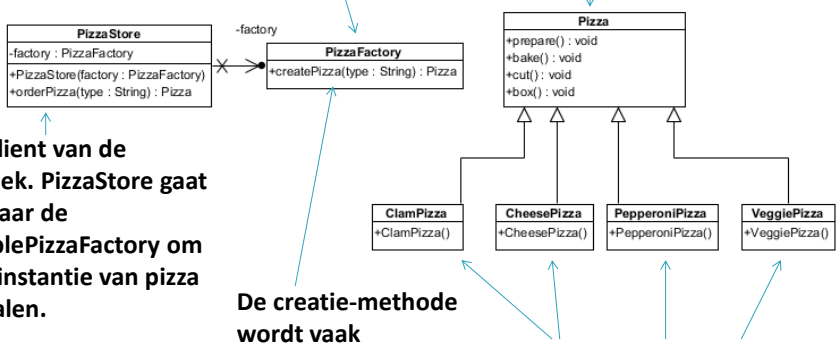

De klasse PizzaStore opnieuw bewerken

```
public class PizzaStore {

    private SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        if (pizza != null) {
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();
        }
        return pizza;
    }
}
```

Definitie van de Simple Factory

De **FABRIEK** waar we pizza's maken. Dit zou het enige deel moeten zijn in de applicatie dat naar de concrete Pizza klassen refereert

Het **PRODUCT** van de fabriek: pizza!



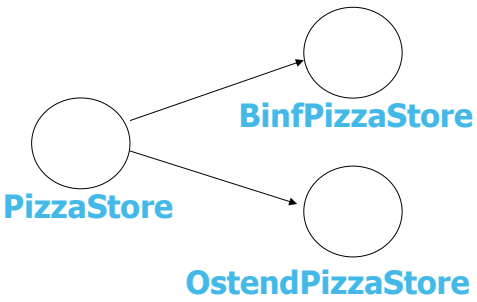
De client van de fabriek. PizzaStore gaat nu naar de SimplePizzaFactory om een instantie van pizza te halen.

De creatie-methode wordt vaak STATISCH gedeclareerd.

De concrete producten.

Franchiseren van een pizzeria

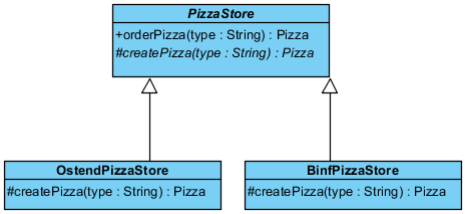
- Hoe zit het met de regionale verschillen? Iedere franchisenemer wil misschien verschillende soorten pizza's aanbieden (BINF, Oostende).



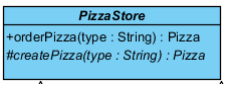

```
BinfPizzaFactory binfFactory
   = new BinfPizzaFactory();
PizzaStore binfStore
  = new PizzaStore (binfFactory );
binfStore.order("Veggie");

OstendPizzaFactory ostendFactory
  = new OstendPizzaFactory();
PizzaStore ostendStore
   = new PizzaStore (ostendFactory );
ostendStore.order("Veggie");
```

Franchiseren van een pizzeria

- SimpleFactory-idee is op de markt getest, maar franchisenemers gebruiken wel jouw fabriek, maar passen ook hun eigen huisprocedures toe voor de rest van het proces. Ze bakten de pizza's iets anders.
- Oplossing : Een framework voor de pizzeria
  - PizzaStore is nu abstracte klasse. Onze 'fabrieksmethode' is een abstracte methode.
  - Laat de subklassen beslissen



Abstracte klasse PizzaStore




```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        if (pizza != null) {
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();
        }
        return pizza;
    }

    protected abstract Pizza createPizza(String type);
}
```

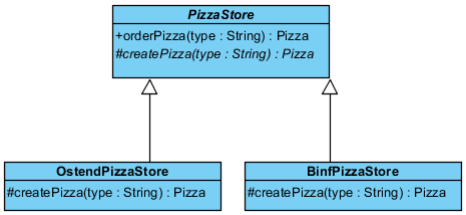
Laat de subklassen beslissen

```
public class BinfPizzaStore extends PizzaStore {

  @Override
  protected Pizza createPizza(String type) {

    switch (type.toLowerCase()) {
      case "cheese":
        return new BinfCheesePizza();
      case "pepperoni":
        return new BinfPepperoniPizza();
      case "clam":
        return new BinfClamPizza();
      case "veggie":
        return new BinfVeggiePizza();
      default:
        return null;
    }
  }
```

Idem voor OstendPizzaStore

Laat de subklassen beslissen.

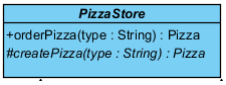
- De subklassen van PizzaStore zijn gewoon subklassen. Hoe kunnen die iets beslissen?
- Leg uit.




```
public Pizza orderPizza(String type) {
    Pizza pizza;

    pizza = createPizza(type);

    if (pizza != null) {
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
    }
    return pizza;
}
```

Laat de subklassen beslissen.

- Verklaring
  - orderPizza() is gedefinieerd in de abstracte PizzaStore, niet in de subklassen. De methode heeft dus geen idee welke subklasse de code feitelijk uitvoert en de pizza maakt.

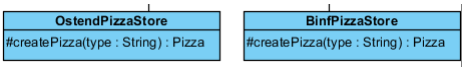


  - orderPizza() roept createPizza() aan om daadwerkelijk een object pizza te krijgen. Maar welk soort pizza krijgt hij? De methode orderPizza() kan dit niet beslissen; zij weet het gewoon niet. Dus wie besluit er dan?

```
public Pizza orderPizza(String type) {
    Pizza pizza;

    pizza = createPizza(type);

    if (pizza != null) {
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
    }
    return pizza;
}
```


Laat de subklassen beslissen.

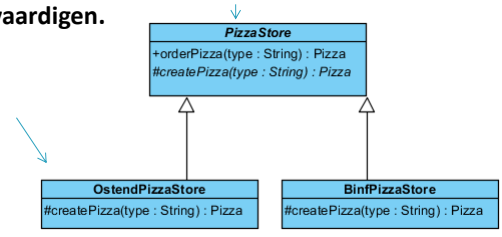
- Verklaring
  - Wanneer orderPizza() createPizza() aanroept, dan wordt één van jouw subklassen aangeroepen om een pizza te maken. Welk soort pizza? Dat is afhankelijk van de pizzeria waar je bestelt, de BinfPizzaStore of de OstendPizzaStore.



  - De subklassen ‘beslissen’ dus niet echt; jij kiest een pizzeria en de subklassen bepalen welke pizza je krijgt

Factory Methode pattern : de creator klasse

De abstracte Creator-klasse. Deze definieert een abstracte fabrieksmethode die door de subklassen geïmplementeerd wordt om producten te vervaardigen.

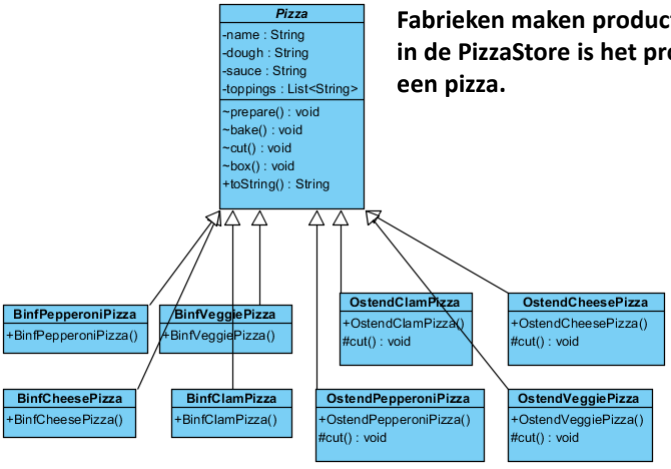


De methode createPizza() is de fabrieksmethode. Deze maakt producten.

Aangezien iedere franchisenemer zijn eigen subklasse van PizzaStore krijgt, heeft hij de vrijheid om zijn eigen stijl pizza's te maken via de implementatie van createPizza().

Klassen die producten maken, heten concrete creators.

Factory Methode pattern : de Product klassen



Fabrieken maken producten en in de PizzaStore is het product een pizza.

Dit zijn concrete producten – allemaal pizza's die door onze pizzeria's worden gemaakt.

Factory Methode pattern : de Product klassen

- PIZZA!

```
public abstract class Pizza {

    private String name;
    private String dough;
    private String sauce;
    private List<String> toppings = new ArrayList<>();

    public void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        toppings.forEach(topping -> System.out.printf("   %s", topping));
    }

    public void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
```

De abstracte klasse levert een standaardinstelling voor het bakken, snijden en verpakken.

… Zie project 'FactoryMethod'

Factory Methode pattern : de Product klassen

- De concrete subklassen

```
public class OstendPepperoniPizza extends Pizza {

    public OstendPepperoniPizza() {
        setName("Ostend Style Pepperoni Pizza");
        setDough("Extra Thick Crust Dough");
        setSauce("Plum Tomato Sauce");

        addTopping("Shredded Mozzarella (
        addTopping("Black Olives");
        addTopping("Spinach");
        addTopping("Eggplant");
        addTopping("Sliced Pepperoni");
    }

    @Override
    public void cut() {
        System.out.println("Cutting the pizz
```

```
public class BinfPepperoniPizza extends Pizza {

    public BinfPepperoniPizza() {
        setName("BINF Style Pepperoni Pizza");
        setDough("Thin Crust Dough");
        setSauce("Marinara Sauce");

        addTopping("Grated Reggiano Cheese");
        addTopping("Sliced Pepperoni");
        addTopping("Garlic");
        addTopping("Onion");
```

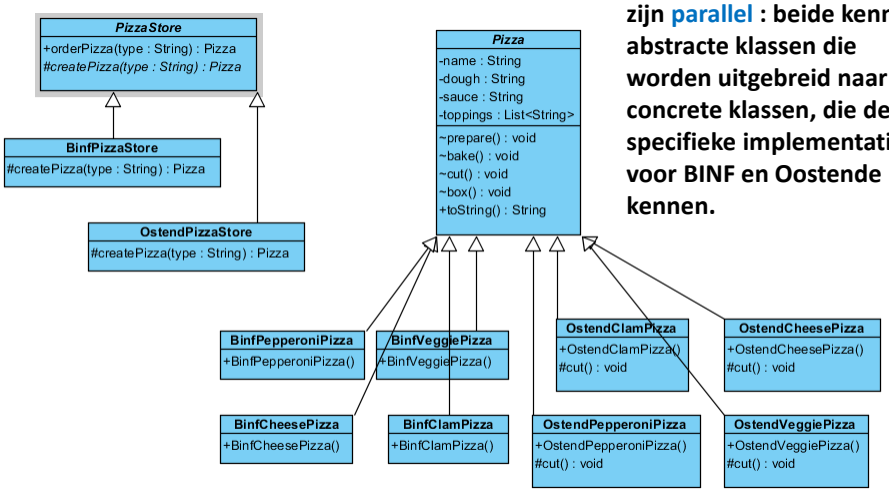
En dan nu …

```
import domein.OstendPizzaStore;
import domein.Pizza;
import domein.PizzaStore;

public class PizzaTestDrive {

  public static void main(String[] args) {
    PizzaStore binfStore = new BinfPizzaStore();
    PizzaStore ostendStore = new OstendPizzaStore();

    Pizza pizza = binfStore.orderPizza("cheese");
    System.out.println("Ethan ordered a " + pizza.ge

    pizza = ostendStore.orderPizza("cheese");
    System.out.println("Joel ordered a " + pizza.getN

    pizza = binfStore.orderPizza("clam");
    System.out.println("Ethan ordered a " + pizza.ge
```

```
Preparing BINF Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Reggiano CheeseBake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a BINF Style Sauce and Cheese Pizza

Preparing Ostend Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella CheeseBake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Ostend Style Deep Dish Cheese Pizza
```

HoGent

Factory Method Pattern



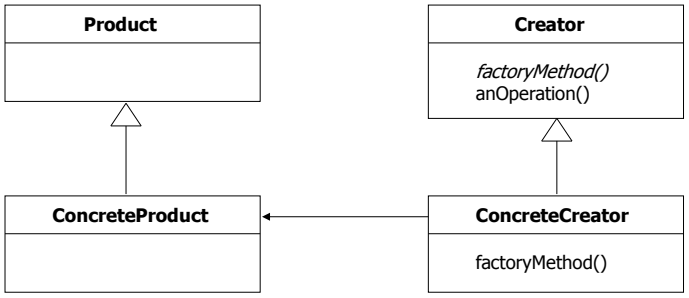
Deze klassen-hiërarchiën zijn parallel : beide kennen abstracte klassen die worden uitgebreid naar concrete klassen, die de specifieke implementaties voor BINF en Oostende kennen.

HoGent

20

Factory Method Pattern

Het Factory Method Pattern definieert een interface voor het creëren van een object, maar laat de subklassen beslissen welke klasse er geïnstantieerd wordt. De Factory Method draagt de instanties over aan de subklassen

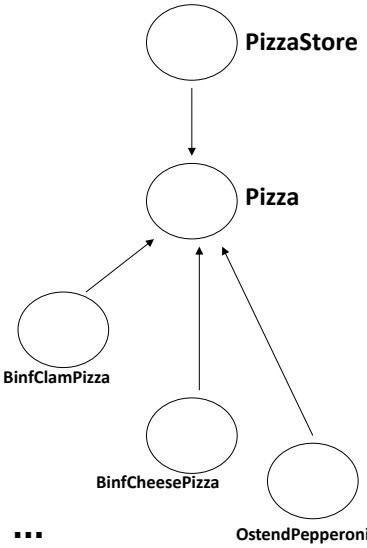


Het Dependency Inversion-principe

Wees afhankelijk van abstracties.
Wees niet afhankelijk van concrete klassen

- Het principe suggereert dat onze high-levelcomponenten niet afhankelijk mogen zijn van onze low-levelcomponenten. Beiden zouden moeten afhangen van abstracties.

Het principe toepassen



PizzaStore is nu alleen afhankelijk van de abstracte klasse Pizza

Pizza is een abstracte klasse … een abstractie

De concrete pizza's zijn nu ook afhankelijk van de abstracte Pizza, omdat ze de Pizza-interface in de abstracte klasse Pizza implementeren (woord 'interface' -> in de algemene zin)

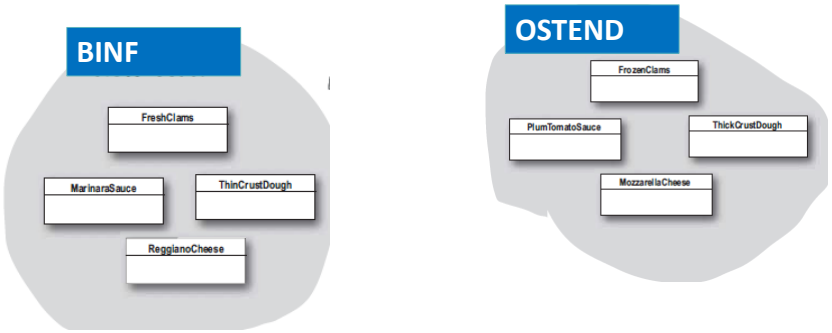
Terug naar de PizzaStore

- Uitbreiding:
  - Elke franchisenemer gebruikt andere ingrediënten.
  - BINF gebruikt een verzameling ingrediënten en Oostende een andere verzameling.

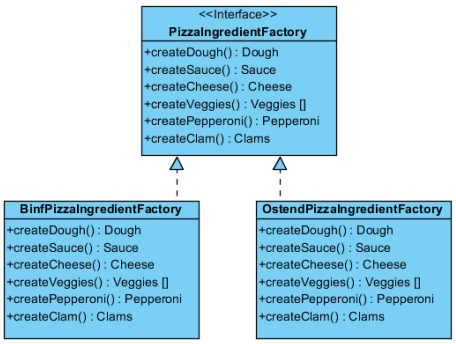
Dezelfde product families (deeg, kaas,…) maar andere implementaties

Terug naar de PizzaStore

- Uitbreiding:
  - Elke franchisenemer gebruikt andere ingrediënten.



Ingrediëntenfabriek bouwen



Voor ieder ingrediënt definiëren we een methode create in de interface.

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

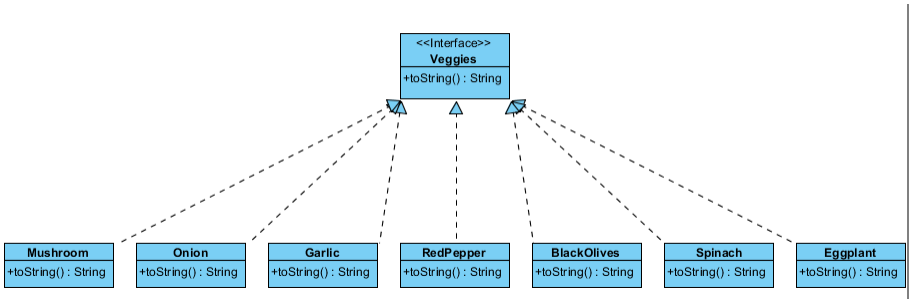
De ingrediëntenfabriek voor BINF bouwen

```
public class BinfPizzaIngredientFactory implements PizzaIngredientFactory {

  public Dough createDough() {
    return new ThinCrustDough();
  }

  public Sauce createSauce() {
    return new MarinaraSauce();
  }

  public Cheese createCheese() {
    return new ReggianoCheese();
  }

  public Veggies[] createVeggies() {
    Veggies veggies[] = {new Garlic(), new Onion(), new Mushroom(),
      new RedPepper()};
    return veggies;
  }
```

Voor ieder ingrediënt in de ingrediëntenreeks, creëren we een BINF-versie.

27

De ingrediënten



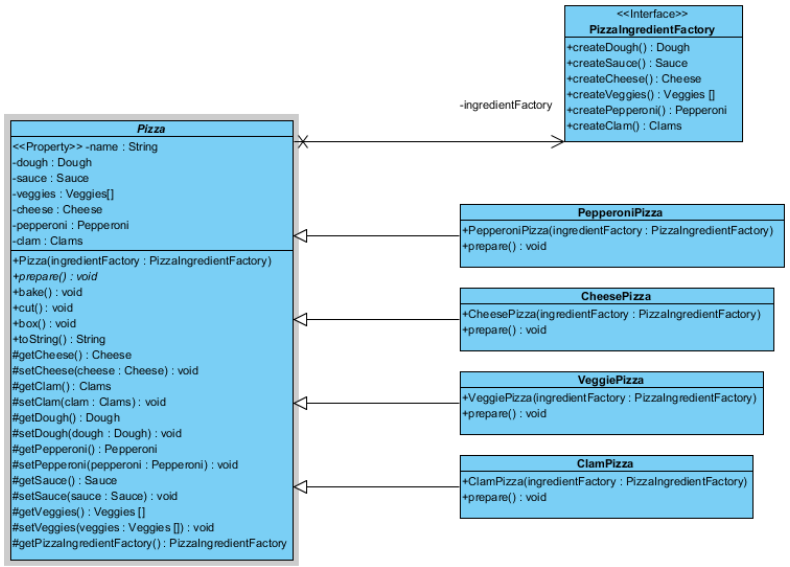

```
public interface Veggies {

  public String toString();
}
```

```
public class RedPepper implements Veggies {

  public String toString() {
    return "Red Pepper";
  }

}
```

HoGent

28

De pizza's opnieuw bewerken



De klasse Pizza

```
public abstract class Pizza {

  private String name;
  private Dough dough;
  private Sauce sauce;
  private Veggies veggies[];
  private Cheese cheese;
  private Pepperoni pepperoni;
  private Clams clam;
  private PizzaIngredientFactory ingredientFactory;
  public Pizza(PizzaIngredientFactory ingredientFactory) {
    this.ingredientFactory = ingredientFactory;
  }
  public abstract void prepare();

  public void bake() {
    System.out.println("Bake for 25 minutes at 350");
  }
```

Ieder pizza kent een lijst ingrediënten die bij de bereiding worden gebruikt.

We hebben een fabriek nodig die ingredienten levert

Methode prepare wordt een abstracte methode. Hier verzamelen we de ingrediënten die voor een pizza nodig zijn en die komen natuurlijk uit de ingrediëntenfabriek.

De andere methoden veranderen niet, met uitzondering de methode prepare

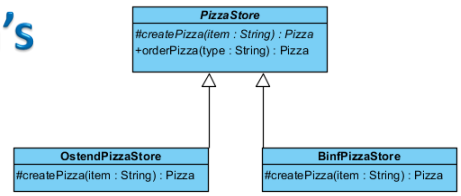
Een concrete klasse

```
public class PepperoniPizza extends Pizza {

  public PepperoniPizza(PizzaIngredientFactory ingredientFactory) {
    super(ingredientFactory);
  }

  public void prepare() {
    System.out.println("Preparing " + getName()
    setDough(getPizzaIngredientFactory().createDough());
    setSauce(getPizzaIngredientFactory().createSauce());
    setCheese(getPizzaIngredientFactory().createCheese());
    setVeggies(getPizzaIngredientFactory().createVeggies());
    setPepperoni(getPizzaIngredientFactory().createPepperoni());
  }
}
```

PepperoniPizza kent ook een ingrediëntfabriek

De methode prepare() stapt door de vervaardiging van een pepperonipizza, en iedere keer dat een ingrediënt nodig is, vraagt ze aan de fabriek om het te maken.

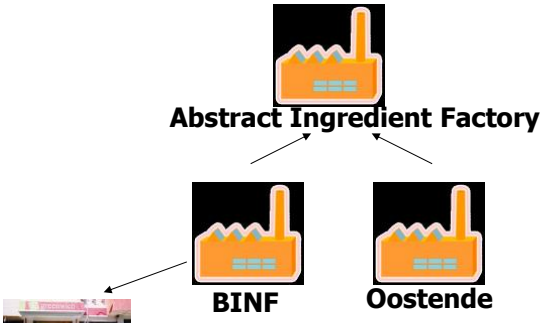
Terug naar pizzeria's




```
public class BinfPizzaStore extends PizzaStore {

  protected Pizza createPizza(String item) {
    Pizza pizza = null;
    PizzaIngredientFactory ingredientFactory = new BinfPizzaIngredientFactory();

    switch (item.toLowerCase()) {
      case "cheese":
        pizza = new CheesePizza(ingredientFactory);
        pizza.setName("BINF Style Cheese Pizza");
        break;

      case "veggie":
        pizza = new VeggiePizza(ingredientFactory);
        pizza.setName("BINF Style Veggie Pizza");
        break;
    //....
```

Zie project 'AbstractFactory'

Abstract Factory Pattern



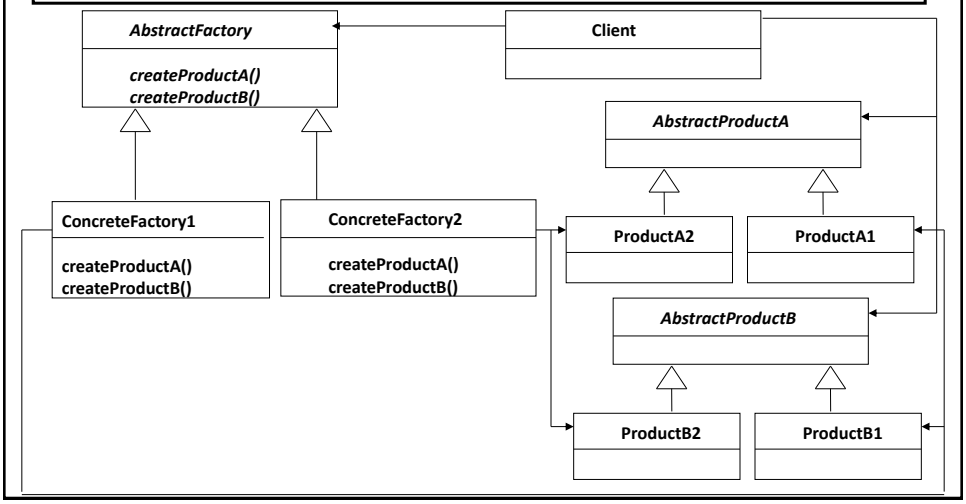
Een abstract Factory levert een interface voor een reeks producten. In ons geval alle dingen die nodig zijn om een pizza te maken: deeg, saus, kaas, vleeswaren en groenten.

We schrijven onze code zodanig dat deze de fabriek gebruikt voor het maken van producten. Door een verscheidenheid aan fabrieken krijgen we een verscheidenheid aan implementaties voor de producten. Maar onze clientcode blijft hetzelfde.

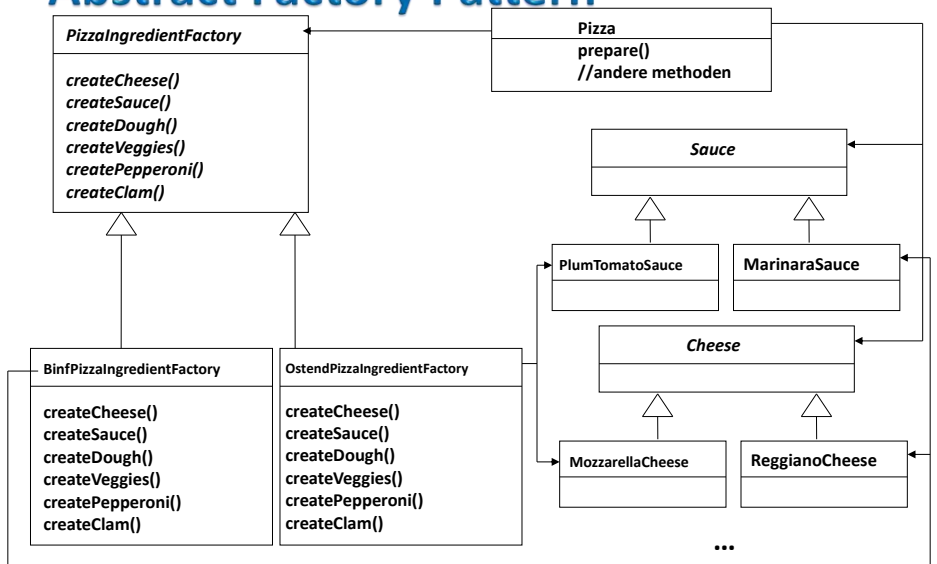
Pizza gemaakt van de ingrediënten vervaardigd door een concrete fabriek.

Abstract Factory Pattern

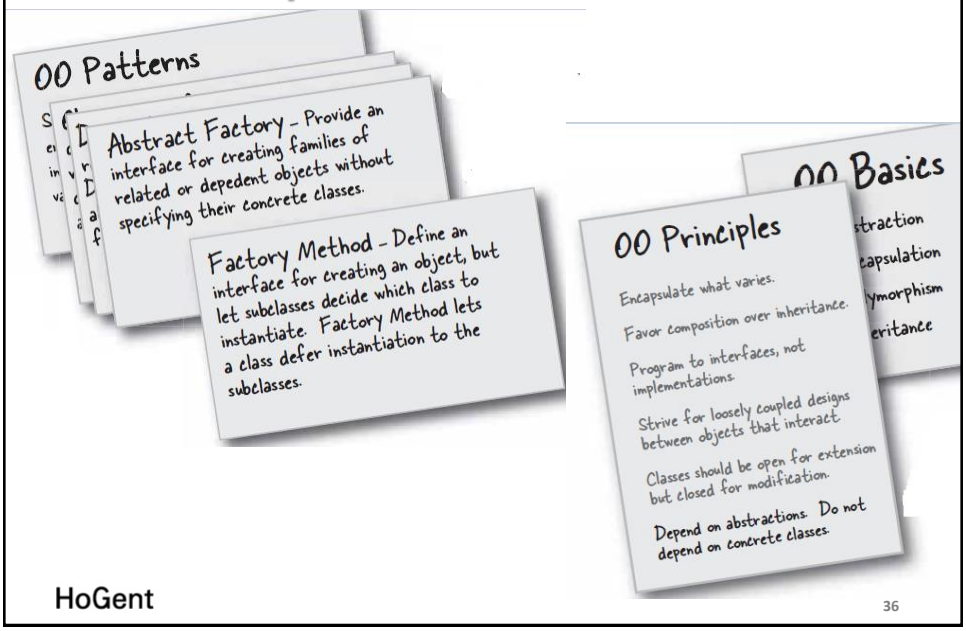
Het Abstract Factory Pattern levert een interface voor de vervaardiging van reeksen gerelateerde of afhankelijke objecten zonder hun concrete klassen te specificeren



Abstract Factory Pattern



Je ontwerp-toolbox



HoGent

36

Referentielijst

- Eric, F., & Elisabeth, F. (2004). *Head First Design Patterns* (p. 629). O'Reilly Media, Inc.